VZ200 Extended Dos Basic

(draft)

Contents

Introduction

Purpose

The purpose of this document is to capture the hardware and software design of a custom VZ200 cartridge that will:

1. Expand the RAM of a VZ200 or VZ300 to the maximum possible 34k
2. Re-enable the Level II basic commands that are disabled in the standard V2.0 ROM
3. Implement a new SD card based file system to load and save programs and data, providing a hardware alternative to the ageing and rare floppy disk drive

Scope

A custom circuit board will be built to fit inside a standard cartridge case. The board will contain

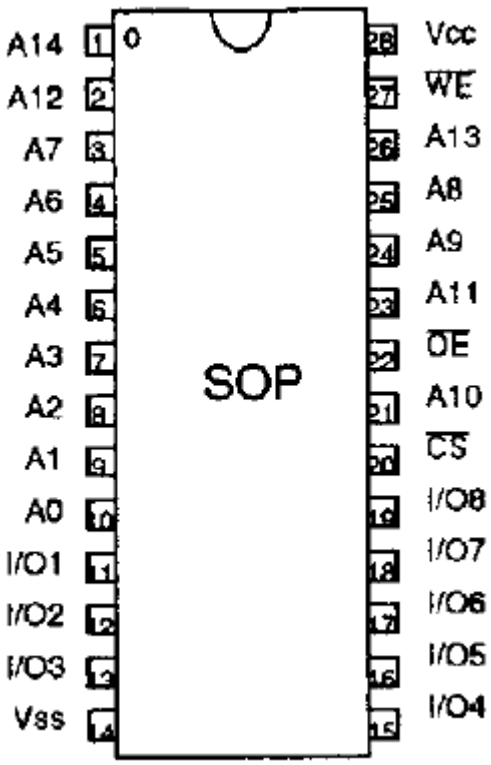
- one 32k static ram (KM62256BLP-8 or equivalent)
- one 16k eprom (CY27C128 or equivalent)
- two address decoders (74LS138)
- one teensy 3.5 microcontroller. This is a powerful Arduino compatible microcontroller with a built-in SD card, 192kb RAM, and digital I/O ports

Design

RAM

The RAM will occupy address range 8000h - FFFFh. This range overlaps the existing internal memory in the VZ200 and VZ300, however it also means the one cartridge will expand both machines to the maximum possible conventional RAM.

The pinout of the 32k static RAM chip is below.



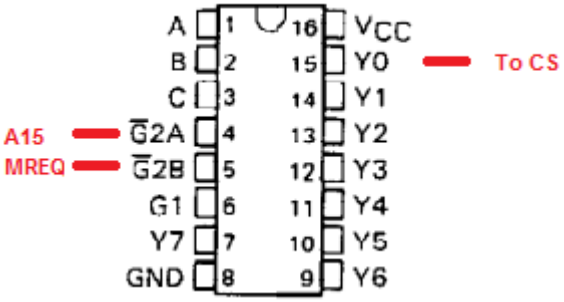
The chip needs to be enabled for addresses 0x8000-0xFFFF. This corresponds to the Z80 A15 address line going high, and the Z80 MREQ line going (active) low. If MREQ is low, the Z80 will be attempting either a memory read or a write, so the RD and WR signals don't need to be checked.

The Z80 RD line will connect to the OE pin

The Z80 WR line will connect to the WE pin

The Z80 address and data lines will connect to the corresponding pins

The chip select (CS) pin will be connected to pin Y0 on the paired 74LS138. The Z80 A15 line will connect to G1. The Z80 MREQ line will connect to G2. The A, B and C inputs will be grounded. This combination will activate the chip.



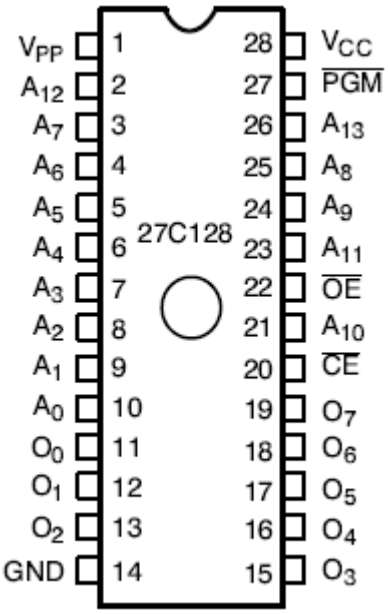
INPUTS					OUTPUTS							
ENABLE		SELECT										
G1	$\overline{G2}$*	C	B	A	Y0	Y1	Y2	Y3	Y4	Y5	Y6	Y7
X	H	X	X	X	H	H	H	H	H	H	H	H
L	X	X	X	X	H	H	H	H	H	H	H	H
H	L	L	L	L	L	H	H	H	H	H	H	H
H	L	L	L	H	H	L	H	H	H	H	H	H
H	L	L	H	L	H	H	L	H	H	H	H	H
H	L	L	H	H	H	H	H	L	H	H	H	H
H	L	H	L	L	H	H	H	H	L	H	H	H
H	L	H	L	H	H	H	H	H	H	L	H	H
H	L	H	H	L	H	H	H	H	H	H	L	H
H	L	H	H	H	H	H	H	H	H	H	H	L

*$\overline{G2} = \overline{G2A} + \overline{G2B}$
H = high level, L = low level, X = irrelevant

EPROM

The EPROM will occupy the first 8k of the 10k total cartridge space - 0x4000 to 0x6000. This simplifies the address decoding and allows for an additional 2k RAM expansion which could be useful as a graphics buffer, or even banked RAM expansion.

The pinout of the 16k eprom is below



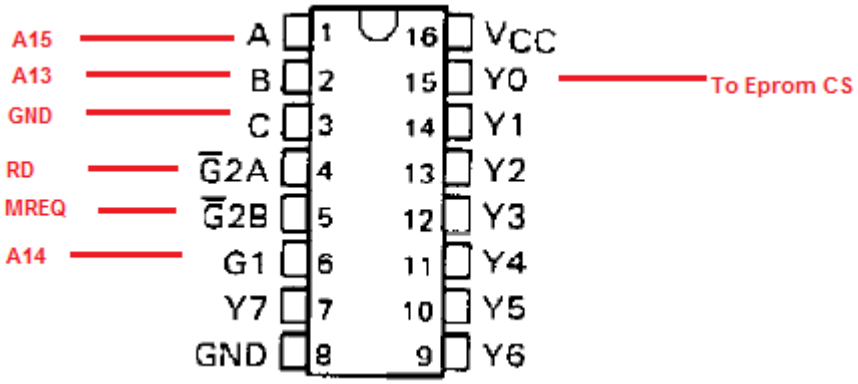
The chip needs to be enabled for addresses 0x4000 to 0x6000. There are three address lines of interest – A15, A14, and A13.

A15 and A13 must be low, and A14 high. MREQ must be low. As this is a read-only eprom, RD must be low.

The Z80 address and data lines will connect to the corresponding pins

The OE (output enable) pin will be tied to CE (chip enable)

The chip enable (CE) pin will be connected to pin Y0 on the paired 74LS138. The Z80 connections are shown below



Microcontroller

The teensy 3.5 has over 40 digital I/O pins available that are 5 volt tolerant, which means they can work with the 5 volt Z80 address and data lines.

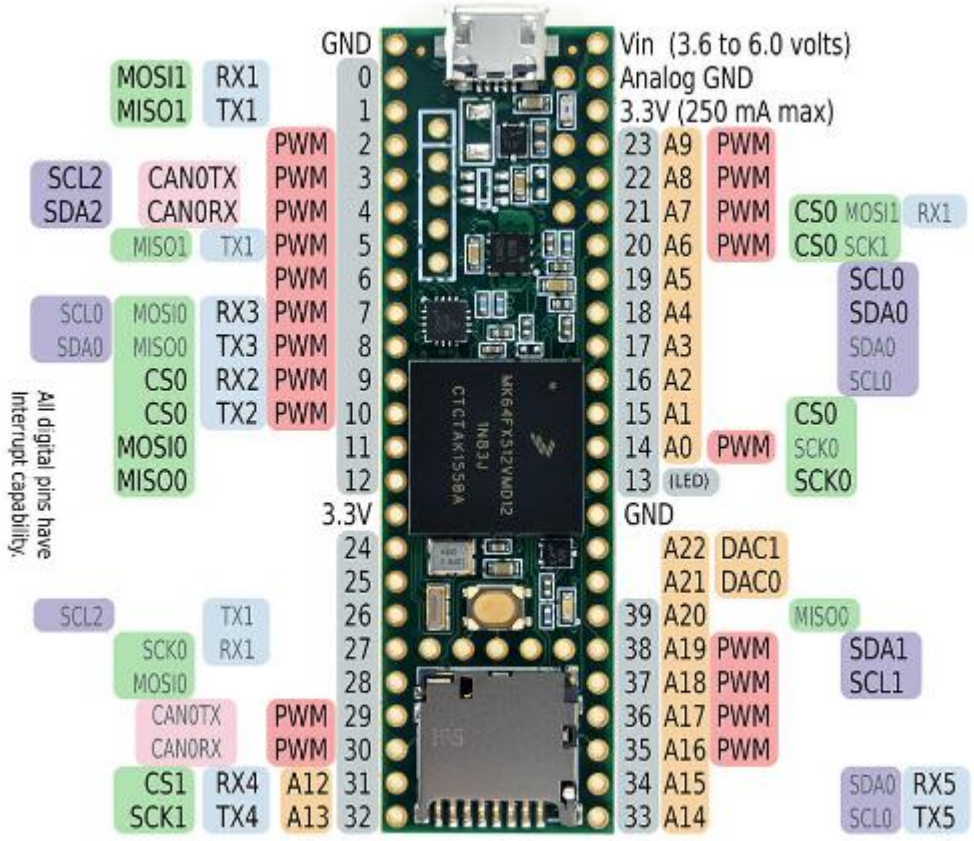
15 pins below are required to interface the Z80 to the teensy for data I/O

- address lines A0-A7
- data lines D7 – D4
- IORQ
- RD
- WR

Note that only 4 data lines are utilised, not all 8. Experiments with the VZ200 showed the data line D0 was always high on a I/O read. Having 4 data lines means the circuit will be simpler, but the data must be sent or received in two passes and bit shifted. Interestingly, the VZ floppy disk interface also only uses 4 data bits.

The exact pin assignment on the microcontroller has not been finalized at the time of writing. The pin selection will be determined according to optimal traces on the circuit board.

The pinout of the teensy is below:



Pins 0 to 12, 24 to 32, and 13 to 23 are available for digital I/O use.

Port range 0x10 to 0x1F will be used to communicate with the microcontroller. This is the same range used in the VZ floppy drive.

The following extended basic commands will be added to allow file operations with the SD card:

BLOAD filename	Loads a binary file to VZ memory at the same address it was saved
BRUN filename	Same as BLOAD, but jumps to the start address
BSAVE filename, start, end	Saves a binary file to the SD card. The start/end addresses are in hex
CD directory	Changes to the names subdirectory on the SD card
CP file1 file2	Creates a copy of file1 named file2

DIR	Displays the directory contents on the screen
LOAD filename	Load a basic program from the SD card
MD directory	Make a directory on the SD card
RD directory	Remove a directory from the SD card
SAVE filename	Save the basic program in memory to the SD card
SNAPL filename	Load a .VZ snapshot file from the SD card
SNAPR filename	Same as SNAPL, but executes the program
SNAPS filename, start, end, filetype	Saves a .VZ snapshot to the SD card. Recognized filetypes are F0 for basic, and F1 for binary.

Each of the commands listed will have an associated “token”. The microcontroller will constantly poll the Z80 IORQ and WR lines. When both are active low, the microcontroller will assert a wait state on the Z80 and sample the data bus. The byte received will identify what command was entered. Firmware on the microcontroller and the VZ cartridge will then process that command.

For example, if the user entered BRUN GALAXON.BIN on the VZ, the following sequence would occur

1. The VZ would send byte 163 to the microcontroller, which is the token for BRUN
2. The microcontroller will receive the byte and jump to processing for BRUN
3. The VZ will send the filename as a sequence of bytes, terminated by 00
4. The microcontroller will receive the bytes and build a string
5. The microcontroller will search for the file on the SD card
6. The microcontroller will send back a one byte response to the VZ: 00 = found, FF = not found
7. If the file was not found, the microcontroller will return to polling the IORQ and WR lines for another command
8. If the file was found, the microcontroller will load the file into local memory. The first two bytes will contain the start address. The microcontroller will send the 2 byte start address, the 2 byte file length, then the file contents to the VZ
9. The VZ will display a ? FILE NOT FOUND error if the response byte was FF. Otherwise it will read the first two bytes to build the start address, the next two bytes for the file length, then read the program bytes.

The VZ and microcontroller run at different speeds, so the transfer of bytes needs to be synchronized.

The microcontroller will have I/O pins wired to the Z80 IORQ, RD and WR lines. If an IORQ is detected, the microcontroller will assert a WAIT state on the Z80, sample or write to the data bus, then release the WAIT state.

To test the above technique, the VZ game “Ace of Aces” was loaded from SD card into a VZ300

Below are the low level Z80 routines to transfer data

```
ReadByteFromMicrocontroller:
;
; The microcontroller has 4 I/O pins wired to the the Z80's 4 most significant
; data pins D7-D4. The microcontroller will send a byte using two 4 bit outs.
; This method will reconstruct the two nibbles into register A
; Uses register C as temporary storage, so save prior to call if needed
;
    IN      A, (10h)    ; get 4 low bits in D7-D4
    AND     11110000b   ; mask unwanted bits
    RRCA                ; convert to low nibble
    RRCA
    RRCA
    RRCA
    LD      C, A        ; save in Reg C
    IN      A, (10h)    ; get 4 high bits in D7-D4
    AND     11110000b   ; mask unwanted bits
    OR      C           ; reconstruct byte
    RET                 ; Reg A contains byte
```

```
WriteByteToMicrocontroller:
;
; Write the contents of Register A to the microcontroller in two passes
; using D7-D4
;
    LD      C, A        ; save contents of Reg A
    AND     00001111b   ; get low 4 bits
    RLCA                ; move to D7-D4
    RLCA
    RLCA
    RLCA
    OUT     (10h), A    ; send low 4 bits to controller
    LD      A, C        ; restore original contents
    OUT     (10h), A    ; send high 4 bits to controller
    RET
```

VZ200 Extended Basic

The VZ200 ROM was copied from the TRS-80 model 1, with modifications to work with VZ specific hardware, eg video display, keyboard, speaker and cassette. However many of the ROM routines, such as clearing the screen, were kept intact and located at the same rom address. Possibly for licensing reasons, the VZ200 only had Level 1 basic commands enabled. The Level 2 commands, while still present in the ROM, were made unrecognizable to the interpreter. For example, typing AUTO would generate a syntax error even though the implementation of this command is still present in the ROM.

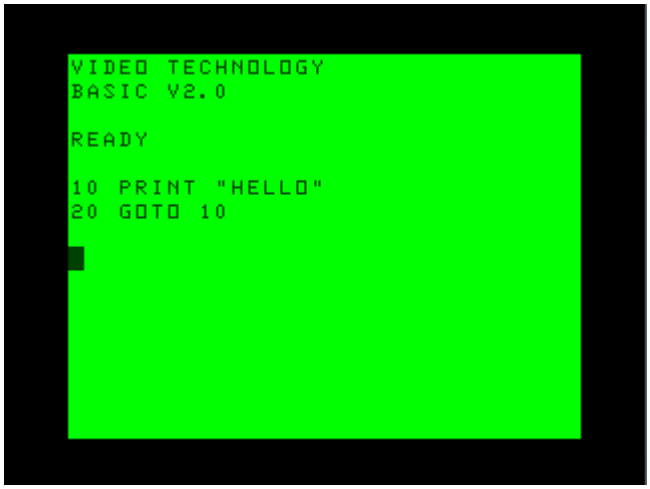
Under the hood – tokenization

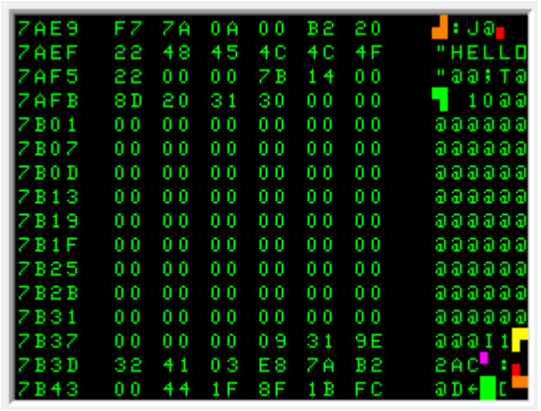
On the VZ200, when the user enters a command, one of the first things the ROM will check is if a line number was entered. A line number will store the commands as part of a program. No line number will be treated as a direct command, to be executed immediately.

Regardless of whether the statement(s) is a direct command or part of a program, the interpreter will scan the input buffer looking for recognized commands. When a recognized command like PRINT is found, the word will be replaced by a single byte, or "token". This serves two purposes - it saves memory, and provides a lookup to the address of the actual routine in ROM

Example 1 – Internal commands:

A VZ200 basic program will normally start at address 7AE9h. When the program below is entered, the memory looks like this…





- The first two bytes reference memory location 7AF7h, which points to the END of the current program line
- The next two bytes, 0A and 00, are the 16 bit line number, which is “10”
- The next byte is B2. This is the token for PRINT. We can see the basic interpreter has replaced the word PRINT with a one byte token
- The next byte is 20, which is an ascii space. The next sequence of bytes contain the “HELLO” message
- The program line is terminated with a zero byte

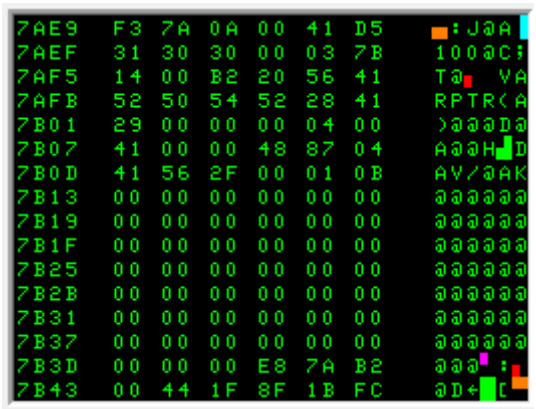
The second line starts with two bytes referencing memory location 7B00h, which points to the end of this program line. The next two bytes are the line number, which is “20”. The next byte, 8D, is the token for GOTO. The following bytes are the ascii representation for a space, the “10” line number and a zero byte for the end of line. The final zero byte signals the end of the program.

If a new line number “15” was entered, the ROM would move subsequent program lines down in memory and slot the new line into memory between lines 10 and 20.

So even before the program has been run, the ROM has done a lot of work tokenizing commands and shuffling memory in preparation for use.

Example 2 - Disabled commands:

If the following program is entered on a stock VZ200, the VARPTR command is not recognized, and therefore not tokenized. Running this will result in an error on line 20



Inspecting the memory from 7AF7, the PRINT command has been converted to token B2, as expected. The VARPTR command was not tokenized, and still exists as a string literal.

However VARPTR is supported by the VZ rom, its' token is C0. This token can be POKE'd back into the program memory, and the rest of the characters cleared with spaces…

```
VIDEO TECHNOLOGY
BASIC V2.0

READY

10 A=100
20 PRINT VARPTR(A)

POKE 31481,192:FOR I=0 TO 4:POKE
 31482+I,32:NEXT
READY
```

The memory contents now contain

```
7AE9  F3 7A 0A 00 41 D5   :J@A
7AEF  31 30 30 00 03 7B   100@C;
7AF5  14 00 B2 20 C0 20   T@
7AFB  20 20 20 20 28 41       (A
7B01  29 00 00 00 04 00   )@@@D@
7B07  49 00 00 20 83 0A   I@@  J
7B0D  00 49 00 00 40 83   @I@@@
7B13  04 41 56 2F 00 01   DAV/@A
7B19  0B 00 00 00 00 00   K@@@@@
7B1F  00 00 00 00 00 00   @@@@@@
7B25  00 00 00 00 00 00   @@@@@@
7B2B  00 00 00 00 00 00   @@@@@@
7B31  00 00 00 00 00 00   @@@@@@
7B37  00 00 00 09 31 9E   @@@I1
7B3D  32 41 03 08 7B B2   2ACH;
7B43  00 44 1F 8F 1B FC   @D
```

```
VIDEO TECHNOLOGY
BASIC V2.0

READY

10 A=100
20 PRINT VARPTR(A)

POKE 31481,192:FOR I=0 TO 4:POKE
 31482+I,32:NEXT
READY
RUN
 31496
READY
```

If the program is run, the interpreter happily works with the replaced token and prints the memory address of variable A

```
READY
LIST
10 A=100
20 PRINT
READY
```

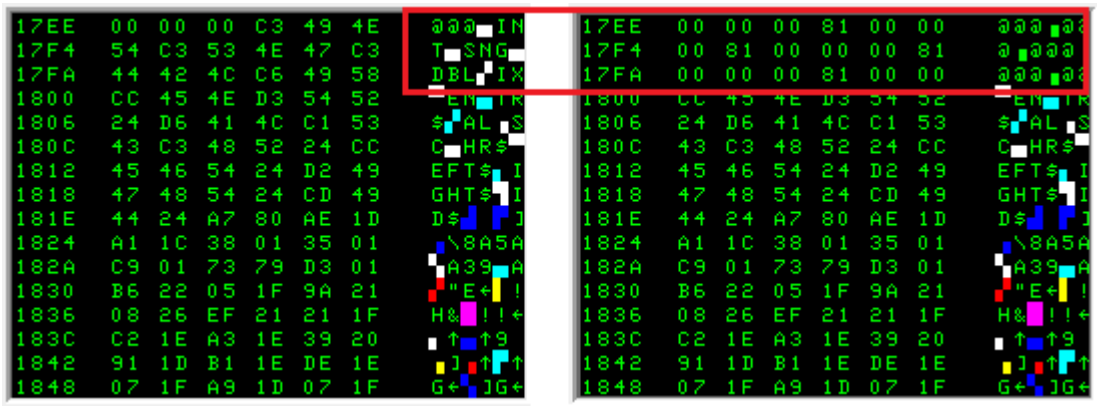
But look what happens when we try to LIST the program – the token is not recognized, so line 20 just contains the PRINT.

The above technique can still be a useful way to get Level 2 Basic commands working with an unmodified VZ200. The only limitation is once the commands have been POKE'd into memory, a LIST will not display them properly.

Restoring the missing commands

There are three options:

1. Replace the stock VZ200 rom with an eprom containing the missing commands. This is the most effective solution, but requires a risky hardware modification. The pictures below show the enhanced V2.1 eprom compared with the stock V2.0 rom. The extra commands visible in this screenshot are CINT, CSNG, CDBL and FIX.



2. Write a utility that interacts with the interpreter to recognize the additional commands. This is the safest option, but also requires the utility is loaded first for every program that uses the additional commands. Workable with a disk system, not so much with tape. A resident utility program will also reduce the ram available to the user.

3. Build a cartridge with an eprom containing the utility. This will work as soon as the VZ is switched on and use little or no system RAM. Additional RAM can also be built into the cartridge. It will not be compatible with the disk rom however, so will only be useful for non disk systems.

Note: Options 2 and 3 use the same mechanism to enable the additional commands.

Get your hooks in

The VZ200 rom was built with future expansion in mind. Contained in RAM are a dozen or so "callbacks", where the ROM jumps to a location in the RAM communications area. By default these

just contain a RET (return) instruction, so do nothing. However because these callbacks are in RAM, they can be modified to JP (jump) to a custom routine.

Examples:

- When an interrupt is generated from the video field synch, a callback is triggered. This can and has been used to great effect to create timers, non erasable displays, and smooth animations.
- When a basic program is running and the user hits ctrl+break, a callback is triggered. This can be used to resume execution, instead of giving control back to the user. This effectively "break-proofs" the running program.

The two callbacks of interest here are:

1. 79B2h – called after standard tokenization finishes, for either a direct command or program line
2. 79DFh – called after a program line is de-tokenized by a LIST, before the line is actually displayed

To add additional basic commands, the callback at 79B2h is modified to jump to a custom routine that rescans the buffer looking for the additional commands, and if found replaces them with a token

To allow these commands to be listed, the callback at 79DFh is modified to jump to a custom routine that rescans the buffer looking for the tokens, and if found replaces them with the command

Reading disabled commands

Below is the list of Level 2 commands in the VZ200 basic rom that are disabled by default, and their token.

token	command	description
134	RANDOM	Reseed random number generator
153	DEFINT	Define a variable as integer
154	DEFSNG	Define a variable as single precision
155	DEFDBL	Define a variable as double precision

159	RESUME	Continue program after error handling
161	ON	Used with GOTO for program flow
182	DELETE	Delete program lines
183	AUTO	Automatic numbering of program lines
192	VARPTR	Returns address of variable
194	ERL	Returns line number where error occurred
195	ERR	Returns error code
200	MEM	Returns free memory in bytes
218	FRE	Returns amount of memory available for variables of the same type
239	CINT	Cast to 16 bit integer
240	CSNG	Cast to single precision
241	CDBL	Cast to double precision
196	STRING$	Returns string of length N

These commands all have existing routines in ROM. Once the commands are tokenized, the basic interpreter will know how to execute them.

In plain English, the callback must jump to a custom routine to rescan the input buffer for the above words, and if found, replace those words with its token.

Adding brand new commands

The VZ rom also contains “spare” tokens, intended for use with a disk expansion. These tokens point to locations in RAM that can be modifed to jump to whatever implementation is required.

This design is inherited from the TRS-80, and the VZ disk rom also uses these tokens for its internal commands.

The list of “spare” tokens is below:

token	command	Callback address (decimal)
133	CMD	31091
162	OPEN	31097
163	FIELD	31100
164	GET	31103
165	PUT	31106
166	CLOSE	31109
167	LOAD	31112
168	MERGE	31115
169	NAME	31118
170	KILL	31121
172	LSET	31127
173	RSET	31130
174	SAVE	31136
176	DEF	31067
230	CVI	31058
231	CVS	31064
232	CVD	31070
233	EOF	31073
234	LOC	31076

There is some limitations on their use however:

1. Tokens up to 187 must begin a basic statement if they are used at all
2. Tokens from 230 onwards must NOT begin a basic statement. These are intended to be used as functions, not statements.

Some commands can run standalone, or contain arguments. An example of this is LIST.

Some commands must contain one or more arguments. An example of this is POKE.

The command may require its arguments to be of a certain type and range. For example the POKE command expects a 16 bit integer followed by an 8 bit integer. These can be parsed as numeric literals, eg POKE 30744,1 or variables, eg POKE A, V

The SOUND command expects two integer values in the range of 1 to 31 for the frequency, and 1 to 9 for the duration.

A good way to build a new command, is to examine an existing command that takes the same arguments.

Example – POKE

The POKE routine address is located at 2CB1h. Below is the disassembly

```
2CB1  CD 02 2B    CALL 2B02h
2CB4  D5          PUSH DE
2CB5  CF          RST 08h
2CB6  2C          INC L
2CB7  CD 1C 2B    CALL 2B1Ch
2CBA  D1          POP DE
2CBB  12          LD (DE),A
2CBC  C9          RET
```

The call to 2B02h evaluates the first argument expression and puts the 16 bit integer result into register DE, which is then saved on the stack. The next three commands evaluate the second argument expression and puts the 8 bit integer result into register A. The address is finally POP'd off the stack and restored into DE, with the address being loaded with register A.

New command – BEEP

BEEP will take exactly the same arguments as POKE, that is, a 16 bit integer and an 8 bit integer. Below is the implementation:

```
BEEP_entry:
    CALL 2B02h          ; evaluate integer expression
    PUSH DE             ; save in DE
    RST 08h             ; get next argument
    INC L               ; evaluate 8 bit expression
    CALL 2B1Ch          ; result in A
    POP DE
    PUSH    BC          ; save basic pointers destroyed by sound driver
    PUSH    DE
    PUSH    HL
    PUSH    DE
    POP     HL          ; put the 16 bit integer into HL
    LD      B, 0        ; put the 8 bit expression into BC
    LD      C, A
    DI
    CALL    345Ch       ; call the sound driver
    EI
    POP     HL          ; restore registers and exit
    POP     DE
    POP     BC
    RET
```

Which is almost identical to POKE. The following code will tie this routine to token 133

```
LD          (31091),A               ; address for token 133
LD          HL,BEEP_entry
LD          (31092), HL
```

To rescan the buffer for the “BEEP” command, the callback at 79B2 is modifed to point to a custom routine

```
LD          A, 0C3h     ; JP instruction
LD          HL, tokenise_extended_commands
LD          (79B2h), A
LD          (79B3h), HL
```

And finally the code to rescan the buffer

```
;==============================================================================
tokenise_extended_commands:
        ;
        ; start of routine to scan for additional commands
        ; save all registers on entry. On exit, the C register
        ; needs to contain the new line length
        ;
        PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL
        PUSH    IX

        LD              A, C
        LD              (7200h), A                      ; TODO - allocate some memory for this.
Save original line length
        XOR             A
        LD              (7201h), A                      ; assume token will not be found, set
command length to zero

        LD              IX, commands_list       ; start of new commands list
        LD              C, 00                           ; each command has a token offset
        INC             HL
check:
        LD              A, (IX+00)                      ; does character in buffer match the
        CP              (HL)                            ; start of a command
        JR              NZ, no_match            ; no, search for next command in list
        LD              B,00                            ; otherwise check all characters match
        PUSH    IX
        PUSH    HL
        CALL    keep_matching
        LD              A,(IX+00)                       ; @ character in IX means command matched
        POP             HL
        POP             IX
        CP              40h                                     ; @ ?
        JR              NZ, no_match            ; no match, so look for next command
        ;
```

```
        ; full command found, length of command in register B. Get token offset
        ; and replace in buffer
        ;
        CALL    tokenise
no_match:
        CALL    next_command          ; locate end of current command, then
        INC             IX                                      ;  position  to  start  of  next
command
        LD              A,(IX+00)
        CP              0FFh                            ;  is  it  terminating  character/no  more
commands?
        JR              NZ, check                       ; no - keep looking for commands
        LD              C, 00                           ; otherwise reset token offset
        LD              IX,commands_list        ; and check for all commands again
        INC             HL                                      ; from the next character in the
buffer
        LD              A,(HL)
        OR              A                                       ;  input  buffer  terminates  zero
bytes
        JR              NZ,check                        ; so if not zero, keep checking buffer
        POP             IX
        POP             HL
        POP             DE

        LD              A, (7201h)                      ; get found command length (or zero if
not found)
        LD              B, A
        LD              A, (7200h)                      ; get original line length
        SUB     B

        POP             BC                                      ; restore original value
        LD              C, A                            ; adjust for new line length
        INC             C                                       ; for 1 byte token
        POP             AF

        RET

;===============================================================================
keep_matching:
        ;
        ;       compare the byte pointed to IX with HL. Exit when they no longer match
```

```
        ;       IX references the command list, HL references the input buffer
        ;       On exit, Reg B contains command length
        ;
        INC             HL
        INC             IX
        INC     B
        LD              A, (IX+00)
        CP              (HL)
        JR              Z, keep_matching
        RET
;===============================================================================
next_command:
        ;
        ;       adjust IX to the start of the next command string
        ;       each command is terminated by 40h, ie "@"
        ;       On exit C register contains the token offset
        ;
        INC             IX
        LD              A,(IX+00)
        CP              40h
        JR              NZ, next_command
        INC             C                                       ; token offset
        RET
;===============================================================================
tokenise:
        ;
        ; convert the command string in the buffer to its token.
        ; eg "BLOAD" will become 162.
        ; On entry HL points to first character of command in buffer
        ;       B contains command length
        ;       C contains token offset
        ;
        ; Need to preserve HL so calling routine can maintain position in
        ; input buffer
        ;
        PUSH    HL

        LD              A, B
        LD              (7201h), A                              ; save command length
```

```
        LD              DE, token_list                  ; point to list of tokens
        LD              A, C                                    ; get token offset
        ADD             A, E                                    ; Adjust DE to point to
        LD              E,A                                     ; right token
        LD              A,(DE)                                  ; get the token
        LD              (HL), A                                 ; and replace first character of
command

        PUSH    HL
        POP             DE
        INC             DE                              ; DE = next byte after token

        LD              C, B                    ; get length in low byte
        LD              B, 00                   ; clear high byte
        ADD             HL, BC                  ; HL now points to first byte after command
string

        ; C register contains command length.

        LD              A, (7200h)              ; get original line length
        SUB     C
        LD              C, A
        LD              B, 0
        LDIR

        POP             HL
        RET

        .org 4400h                      ; needs to be on boundary
commands_list:
        .text "BLOAD@BRUN@BSAVE@CHDIR@COPY@DIR@LOAD@MKDIR@RMDIR@SAVE@SNAPL@SNAPR@"
        .text "SNAPS@STATUS@BEEP@"
        ; existing disabled commands
        .text "AU"
        .db 189         ; token for "TO" to complete "AUTO"
        .text "@"
        .text "R"
        .db 210         ; token for "AND" to continue "RANDOM"
        .text "OM@"
```

```
        .text "DE"
        .db 140         ; token for "LET" to continue "DELETE"
        .text "E@"
        .text "VARPTR@MEM@DEF"
        .db 216         ; token for "INT" to complete "DEFINT"
        .text "@DEFSNG@DEFDBL@ON@RESUME@SYSTEM@"
        .text "ERL@ERR@STRING$@INSTR@MKI$@MKS$@MKD$@CINT@CSNG@CDBL@FIX@"
        .db 0FFh        ; terminating character
token_list:
        ; new sd disk commands

        .db 162         ; BLOAD
        .db 163         ; BRUN
        .db 164         ; BSAVE
        .db 165         ; CHDIR
        .db 166         ; COPY
        .db 167         ; DIR
        .db 168         ; LOAD
        .db 169         ; MKDIR
        .db 170         ; RMDIR
        .db 171         ; SAVE
        .db 172         ; SNAPL
        .db 173         ; SNAPR
        .db 230         ; SNAPS
        .db 231         ; STATUS

        .db 133         ; BEEP

        ; existing disabled commands

        .db 183         ; AUTO
        .db 134         ; RANDOM
        .db 182         ; DELETE
        .db 192         ; VARPTR
        .db 200         ; MEM
        .db 153         ; DEFINT
        .db 154         ; DEFSNG
        .db 155         ; DEFDBL
        .db 161         ; ON
```

```
        .db 159         ; RESUME
        .db 174         ; SYSTEM
        .db 194         ; ERL
        .db 195         ; ERR
        .db 196         ; STRING$
        .db 197         ; INSTR
        .db 236         ; MKI$
        .db 237         ; MKS$
        .db 238         ; MKD$
        .db 239         ; CINT
        .db 240         ; CSNG
        .db 241         ; CDBL
        .db 242         ; FIX

        .db 0FFh        ; terminating character
file_not_found_msg:
        .text "?FILE NOT FOUND ERROR"
        .db 00
sddos_msg:
        .text "EXTENDED BASIC V2.1"
        .db 0Dh
        .db 0Dh
        .db 00
```